Analytical Cache Modeling and Tilesize Optimization for Tensor Contractions

Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, Ponnuswamy Sadayappan

Analytical Cache Modeling and Tilesize Optimization for Tensor Contractions

Rui Li
University of Utah
lirui@cs.utah.edu

Aravind Sukumaran-Rajam
Ohio State University
sukumaranrajam.1@osu.edu

Richard Veras
Louisiana State University
rveras@lsu.edu

Tze Meng Low
Carnegie Mellon University
lowt@andrew.cmu.edu

Fabrice Rastello
French Institute for Research in Computer Science and Automation (INRIA)
fabrice.Rastello@inria.fr

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

P. Sadayappan
University of Utah
saday@cs.utah.edu

ABSTRACT

Data movement between processor and memory hierarchy is a fundamental bottleneck that limits the performance of many applications on modern computer architectures. Tiling and loop permutation are key techniques for improving data locality. However, selecting effective tile-sizes and loop permutations is particularly challenging for tensor contractions due to the large number of loops. Even state-of-the-art compilers usually produce sub-optimal tile-sizes and loop permutations, as they rely on naïve cost models. In this paper we provide an analytical model based approach to multi-level tile size optimization and permutation selection for tensor contractions. Our experimental results show that this approach achieves comparable or better performance than state-of-the-art frameworks and libraries for tensor contractions.

KEYWORDS

tensor contraction, domain-specific compiler optimization, performance modeling, model-driven design-space exploration

1 INTRODUCTION

A tensor contraction is a higher-dimensional generalization of matrix-matrix multiplication. Tensor contractions represent the computationally dominant component of many applications in computational science and machine learning. Consider the following contraction from the CCSD(T) [5] method in computational chemistry, where two 4D tensors are contracted to produce a 6D tensor:

$$C[a,b,c,i,j,k] = A[i,b,a,l] * B[l,c,j,k] \quad (1)$$

It represents the computation:

$$C[a,b,c,d,i,j,k] = \sum_{l} A[i,b,a,l] * B[l,c,j,k]$$

Typically, the sizes of tensors in large-scale calculations vastly exceed cache capacity, thus tiling is a critical loop transformation for efficient implementation of tensor contractions. However, the number of tiling loops is often very large, and much larger is the number of possible permutations of the tiling loops. Further, multi-level tiling must be considered, in order to optimize across a multi-level cache hierarchy. Finally, a challenging modeling aspect that has generally been ignored in prior attempts at analytical tile-size optimization is that of inter-tile data reuse – thus for simplicity the assumption is often made that no reuse of data occurs across successive tiles. However, this assumption is avoided in the well known panel-panel scheme [7] for optimal tiling of matrix-matrix multiplication [12], which makes full use of inter-tile data reuse by keeping a slice of the result matrix stationary across execution of successive tiles.

In this paper we address the above challenges and develop an effective analytical approach for selection of tile permutation and tile-size for multi-level tiled execution of tensor contractions.We will show that this approach is broadly applicable, but our primary focus is that of effective tiling of arbitrary tensor contractions, a fundamentally important primitive for many applications in computational and data science. In this section we provide a high-level sketch of the key ideas behind the developed approach to tile optimization.

The computation for the tensor contraction in Eq. 1 can be expressed as a 7-dimensional loop nest, with one loop per unique index. Allowing for any order of accumulation of additive contributions for each result tensor element, all loops of an arbitrary tensor contraction are fully permutable and hence fully tileable with hyper-rectangular tiles. Considering a three-level memory hierarchy, up to three levels of tiling may be appropriate, leading to an explosively large search space with three groups of 7 tiling loops, with 7! possible permutations of the tiling loops within each group, i.e., 1.28×10^{11} possible configurations.

However, this huge space of permuted orders for the tiling loops can be drastically pruned by showing that only the innermost tiling loop within each band can have a significant effect on performance. This reduces the number of evaluated configurations from $(7!)^3$ (1.28×10^{11}) to 7^3, i.e. only 343 cases. We will elaborate later in the paper that this is a consequence

of the fact that each tensor dimension of any tensor is indexed by a distinct loop index in a tensor contraction.

For a given permutation of tiling loops we develop an analytical formulation for the volume of data movement as a set of conditional expressions in terms of parametric tile sizes. A constrained optimization solver is then used to find optimal solutions to the formulated minimization problem of finding multi-level tile sizes that minimize the effective time to transmit the transferred volume of data at the different levels of the storage hierarchy.

This paper makes the following key contributions:

- It presents the first practically effective analytical formulation (to our knowledge) for multi-level tile-size optimization for arbitrary dimensional tensor contractions;
- It provides a solution for the multi-level tile-size optimization problem that uses a standard constrained optimization solver;
- It presents experimental validation of the proposed approach using 36 benchmarks in the TCCG benchmark suite .

The rest of the paper is organized as follows. Section 2 presents an overview of our approach. Section 3 details our data movement model and loop permutation/tile-size selection strategy. Section 4 describes the micro kernel design and Section 5 describes buffering/packing to reduce data movement. Extensive experimental evaluation is shown in Section 6. Related works are presented in Section 7 and Sections 8 concludes the paper.

2 OVERVIEW OF MODELING APPROACH

Similar to the manner in which standard matrix-matrix multiplication can be expressed as a 3-dimensional loop nest, the computation for the tensor contraction in Eq. 1 can be expressed as a 7-dimensional loop nest, with one loop for each of the indices {a,b,c,i,j,k,l}. Since any order of accumulation of additive contributions for each result tensor element is generally considered to be acceptable by application scientists, all loops of an arbitrary tensor contraction are considered fully permutable and hence fully tileable with hyper-rectangular tiles. Considering a three-level memory hierarchy, up to three *levels* of tiling may be appropriate, leading to an explosively large search space with three groups of 7 tiling loops and 7! possible permutations of the tiling loops within each group, i.e., 1.28×10^{11} possible configurations.

Zero/Full Inter-Tile Reuse The following key observation is used to drastically prune the huge configuration search space: For any arbitrary tensor contraction, each dimension of any tensor is indexed by a distinct index from the surrounding perfectly nested loops. For example, the four dimensions of tensor A in the contraction in Eq. 1 are respectively indexed by the four distinct loop indices i, b, a, and l. Hence, a loop index is either an explicit index in a given tensor or is unused in indexing that tensor. For example, the loop index a is an explicit index for A, and C, but is not used to access elements of B. For a tiled code, we call the loops that iterate over tiles as *tiling loops*. For example, for the matrix multiplication in Listing 2 with tiles "i1,j1,k1", the tiling loops are the ones indexed by i2, j2, and k2. The *innermost tiling loop* is the one indexed by k2. Consider again the CCSD(T) example of Eq. 1. If the innermost tiling loop index is a, successive tiles along that tiled index would repeatedly access exactly the same slice of data for B (because a does not at all affect the addressing of B), while completely distinct slices of data would be accessed by successive tiles for A and C (because a is an explicit index for A and C, causing each tile to access a distinct and disjoint range of values for the tensor dimension indexed by it). Assuming that the combined data-footprint of a tile just fills the cache/ scratchpad, we will have full inter-tile data reuse for elements of B, but no data reuse for A and C. This observation will always hold as soon as the available space in cache or scratchpad memory is disjointly partitioned (thus avoiding conflicts) to hold the slices of data accessed in a tile from the three tensors.

Only Innermost Tiling Loop Matters: Tile sizes at each level are generally chosen to be large enough such that the data-footprint of a tile is close to the cache/scratchpad capacity but does not exceed it. In that case, as successive tiles of the innermost tiling loop are executed, the data for tensors not indexed by that loop stays invariant, while the data slices for other tensors will be completely disjoint from those used in the previous tiles. In the example considered, if a is the index corresponding to the innermost tiling loop, the data slices for B would be invariant for successive tiles, while complete replacement of data slices for A and C would occur. A direct consequence is that no inter-tile reuse is possible for A and C, irrespective of the permutation of the outer six tiling loops. Further, any additional reuse for B through outer tiling loops would only have a marginal effect on total data volume. Indeed, a significant degree of reuse is already achieved for B through the innermost tiling loop, implying that the total data movement for B is already much lower than that for A and C.

The significant implication of the above observation is the following: Consider a given level in the memory hierarchy and its corresponding tiling level. Only the choice of the innermost tiling loop affects the total data volume (ignoring second order effects) for all tensors from/to that memory level. In other words, among all possible tiling loop permutations, we only need to consider the different possible choices for innermost tiling loop, and choose any single arbitrary permutation for all surrounding tiling loops. For the tensor contraction example, this reduces the number of evaluated configurations from $(7!)^3$ (1.28×10^{11}) to 7^3, that is, to only 343 cases.

Conditional Analytical Expressions for Data Volume: In the next section, we develop an approach to analytical modeling of the impact of tile sizes on data volume using the example of matrix-matrix multiplication. The key idea here is that for a restricted but important class of dense tensor computations, including arbitrary tensor contractions, all tensor dimensions are indexed by distinct loop iterators. With such computations, the data footprint of a tile with

```
for(int i = 0; i < Ni; i++)
 for(int j = 0; j < Nj; j++)
  for(int k = 0; k < Nk; k++)
    C[i][j] += A[i][k] * B[k][j]
```

Listing 1: Matrix Multiplication

```
// Tile sizes are assumed to be perfect multiples of problem sizes
for(int i2 = 0; i2 < Ni; i2+=Ti1)
 for(int j2 = 0; j2 < Nj; j2+=Tj1)
  for(int k2 = 0; k2 < Nk; k2+=Tk1)
   for(int i1 = 0; i1 < Ti1; i1++)
    for(int j1 = 0; j1 < Tj1; j1++)
     for(int k1 = 0; k1 < Tk1; k1++)
       C[i1+i2][j1+j2]+=
       A[i1+i2][k1+k2]*B[k1+k2][j1+j2]
```

Listing 2: Tiled Matrix Multiplication

respect to any operand tensor is simply the product of tile extents along indices that appear in the indexing of the tensor. An inner-to-outer traversal of the loop structure enables the development of conditional symbolic expressions for total data movement as a function of parametric tile sizes. The conditional analytical expressions are then optimized by use of a non-convex optimization solver to determine optimal tile sizes.

3 ANALYTICAL CACHE MODELING

This section presents a new model based approach for predicting the volume of data movement for tiled tensor contraction. For simplicity, we begin by assuming that the caches are programmable (scratchpad) and that the cache-line size is one word. We also assume that the performance is only limited by the cache bandwidth. Later we will address issues that reflect a real cache.

3.1 Single level cache modeling

Loop tiling (loop blocking) is a widely used technique to improve data locality. Tiling chunks the iteration space into multi-dimensional blocks, which enables better reuse of data in hyper-rectangular slices. Tiled loop iterators corresponding to a loop i are represented by an ordered list of iterators $i_1, i_2, ..., i_{l+1}$, where l represents the tiling level. Iterator i_{l+1} represents the outermost loop and the i_1 represents the innermost loop. $l == 0$ denotes the statement level. The tile sizes corresponding to each iterator are represented using $Ti_1, Ti_2, ..., Ti_{l+1}$. Listings 1 and 2 illustrates this notation using matrix multiplication as an example. Listing 2 corresponds to one level tiling of the i, j, k loops in Listing 1. The tiled i loop is represented using $i1$ and $i2$. $i2$ (inter tile iterator) iterates over different blocks of i and $i1$ (intra tile iterator) iterates within a block.

For a given tensor contraction code with fixed loop structure (loop permutation), and parametric tile size variables, our objective is to model the data movement between the cache and main memory. The cost modelling is illustrated using Listing 2, which shows pseudo code for matrix multiplication. The number of elements in each array is assumed to be much larger than the cache capacity. Let $DF(A, i)$ represent the data footprint of array A corresponding loop to i (number of unique elements of the array A accessed by loop nest starting at i). Let $DM(A, i)$ represent the data movement between cache and the main memory corresponding to array A at loop i. Listing 3 shows the pseudo-code to compute the data movement. At the statement level, only a single element of an array is accessed ($DM(A, 0) == DF(A, 0) == 1$). If an array is indexed by a given loop iterator i then its data footprint (data movement) corresponding to the i loop is equal to the product of data footprint (data movement) corresponding to the immediate inner loop and the number of i loop iterations. For example, array A is indexed by $k1$. Hence, for each $k1$ loop iteration, a distinct element of the array A is accessed. Thus the data footprint for A at $k1$ is the product of $DF(A, 0)$ and $Tk1$ which is equal to $1 \times Tk1$. Similarly, the $DF(B, k1)$ is $k1$. Since the array C is not indexed by $k1$, multiple $k1$ iterations accesses the same C element ($DF(C, k1) == DF(C, 0) == 1$).

The total data movement for an array for loop i is dependent on the data movement for inner loops and the cache capacity. For example, the data movement cost of the $j1$ loop is dependent on the data movement cost of $k1$. Thus $DF(B, j1) == DM(B, j1) == Tk1 * Tj1$ and $DF(C, j1) == DM(C, j1) == 1 * Tj1$. Since array A is not indexed by $j1$, the data footprint of A at $j1$ is the equal to the data footprint at $k1$ ($DF(A, j1) == DF(A, k1)$). However, data movement for A at $j1$ depends on whether the cache capacity has already been exceeded or not. If the data footprint corresponding to all arrays at $k1$ (immediate inner loop) is less than cache capacity ($DF(A, k1) + DF(B, k1) + DF(C, k1) <= CacheCapacity$), we can load A once and reuse it at $j1$ level ($DM(A, j1) == DM(A, k1)$). However, if the data footprint corresponding to all arrays at $k1$ exceeds cache capacity, A has to be loaded multiple times ($DM(A, j1) == DM(A, k1) * Tj1$).

Table 1 shows the method to traverse all the small dimension size branches. It is built from a one level cache hierarchy with one tiling group GEMM. The table lists all possible combinations whether each of the dimension problem sizes can fit into cache. Therefore, for a GEMM problem on one tiling group with three levels of tiling loops, there would be $2^3 = 8$ different combinations to be considered, which are already in the row of the top tiling loop $i2$.

The reason to consider data movement of each combination separately is, if some of the dimension can fully fit in cache, the footprint for tensors using this index would not change. That means it would not start to swap out other data at this level. For example, if Nk can fully fit in cache, $Tk1 == Nk$, then the $Ti1 \times Tk1$ amount of data footprint of A would not be swapped out. As a result, this part of A will starts to get reuse in the loop level $j2$. However in the normal case where NK is very large, $Tk1 < Nk$, accesses of A will

loop	range	tile condition	A	B	C
i2	Ni	Ti1 <Ni, Tj1 <Nj, Tk1 <Nk	Ni x Nk * Nj/Tj1	Nj x Nk * Ni/Ti1	Ni x Nj
		Ti1 <Ni, Tj1 <Nj, Tk1 ==Nk	Ni x Tk1	Nj x Tk1 * Ni/Ti1	Ni x Nj
		Ti1 <Ni, Tj1 == Nj, Tk1 <Nk	Ni x Nk * Nj/Tj1	Tj1 x Nk * Ni/Ti1	Ni x Tj1
		Ti1 <Ni, Tj1 == Nj, Tk1== Nk	Ni x Tk1	Tj1 x Tk1	Ni x Tj1
		Ti1 == Ni, Tj1 <Nj, Tk1 <Nk	Ti1 x Nk * Nj/Tj1	Nj x Nk * Ni/Ti1	Ti1 x Nj
		Ti1 == Ni, Tj1 <Nj, Tk1 ==Nk	Ti1 x Tk1	Nj x Tk1 * Ni/Ti1	Ti1 x Nj
		Ti1 == Ni, Tj1 == Nj, Tk1 <Nk	Ti1 x Nk * Nj/Tj1	Tj1 x Nk * Ni/Ti1	Ti1 x Tj1
		Ti1 == Ni, Tj1 == Nj, Tk1== Nk	Ti1 x Tk1	Tj1 x Tk1	Ti1 x Tj1
j2	Nj	Tj1 <Nj, Tk1 <Nk	Ti1 x Nk * Nj/Tj1	Nj x Nk	Ti1 x Nj
		Tj1 <Nj, Tk1 ==Nk	Ti1 x Tk1	Nj x Tk1	Ti1 x Nj
		Tj1 == Nj, Tk1 <Nk	Ti1 x Nk * Nj/Tj1	Tj1 x Nk	Ti1 x Tj1
		Tj1 == Nj, Tk1== Nk	Ti1 x Tk1	Tj1 x Tk1	Ti1 x Tj1
k2	Nk	Tk1 <Nk	Ti1 x Nk	Ti1 x Nk	Ti1 x Tj1
		Tk1 == Nk	Ti1 x Tk1	Tj1 x Tk1	Ti1 x Tj1
		L1 capacity			
i1	Ti1		Ti1 x Tk1	Tj1 x Tk1	Ti1 x Tj1
j1	Tj1		1 x Tk1	Tj1 x Tk1	1 x Tj1
k1	Tk1		1 x Tk1	1 x Tk1	1 x 1
statement			1 x 1	1 x 1	1 x 1

Table 1: Table for traversing combinations of dimension size

```
for each loop i from bottom to top
 if (i == 0){ // statement level
  for each tensor A
   DM(A,i) = DF(A,i) = 1;
 }
 else{
  for each tensor A if i ∈ indices of A{
   DM(A,i) = DM(A,i-1) * range(i)
   DF(A,i) = DF(A,i-1) * range(i)
  }
  else{
   DF(A,i) = DF(A,i-1)
   if ∑_A DF(A,i-1) < CacheCapacity
    DM(A,i) = DM(A,i-1)
   else
    DM(A,i) = DM(A,i-1) * range(i)
  }
 }
}
```

Listing 3: Algorithm for computing data movement

starts to go over the whole dimension of Nk, and because of LRU replacement policy, the beginning segment of A will be replaced, which makes the reuse in loop $j2$ impossible.

3.2 Multi-Level cache modeling

Most modern processors have multiple levels of cache. The fastest cache (L1-cache) is designed to have high bandwidth but has low capacity. Higher caches such as L2 and L3 have higher capacity than L1 but lower bandwidth. Multi-level tiling is used to take advantage of multiple levels of cache. The data movement model presented in Section 3.1 can be extended to support multiple cache levels. We assume that each loop is tiled one for each cache level. Listing 4 shows 2-level tiled matrix multiplication code for a machine with 2 levels of cache. Similar to Listing 2, the loop iterators $i1, i2$, and $i3$ represents the tiled i loop. The $DF()$ function presented in Section 3.1 is not dependent on number of cache levels, hence it can be directly used. The $DM()$ function is modified to include cache level as a parameter – $DM(A, i, l)$ represents the data movement between memory hierarchy l and $l+1$ for array A corresponding to i loop. Listing 3 can

```
// Tile sizes are assumed to be perfect multiples of problem
    sizes
for(int i3 = 0; i3 < Ni; i3+=Ti2)
 for(int j3 = 0; j3 < Nj; j3+=Tj2)
  for(int k3 = 0; k3 < Nk; k3+=Tk2)
   for(int i2 = 0; i2 < Ti2; i2+=Ti1)
    for(int j2 = 0; j2 < Tj2; j2+=Tj1)
     for(int k2 = 0; k2 < Tk2; k2+=Tk1)
      for(int i1 = 0; i1 < Ti1; i1++)
       for(int j1 = 0; j1 < Tj1; j1++)
        for(int k1 = 0; k1 < Tk1; k1++)
          C[i1+i2+i3][j1+j2+j3] +=
          A[i1+i2+i3][k1+k2+j3]*
          B[k1+k2+k3][j1+j2+j3]
```

Listing 4: Multi-Level tiling for Matrix Multiplication

be adapted for multi-level tiling by changing $DM(A,i)$ to $DM(A,i,l)$. Line 13 should be modified to 'if $\sum_A DF(A, i-1) < CacheCapacity(l)$'.

3.3 Predicting execution time based on data movement

Our model predicts the execution time of a program as the maximum time required to transfer data between different cache levels. This prediction is based on the assumption that the memory/cache bandwidth is the main performance bottleneck. Memory/cache latency could also affect the execution time; however, they can be hidden using prefetching.

Let L denote the number of cache levels, $C_l \mid l \in 1 \text{ to } L$ denote the cache at level l, C_0 denote the compute unit, and C_{L+1} denote the main-memory. Let $BW_l \mid l \in 1 \text{ to } L$ denote the maximum bandwidth of cache at level l and BW_{L+1} denote the maximum main-memory bandwidth. Let $C\_DM(l)$ denote the volume of data transferred between C_l and C_{l-1}. Let $C\_time(l)$ denote the time required to move $C\_DM(l)$ elements between C_l and C_{l-1}. For a given loop permutation $\mathcal{P}$, C_time can be computed as follows

$$C\_time(\mathcal{P}, l) = C\_DM(l)/BW_l \tag{2}$$

The predicted execution time is:

$$TotTime(\mathcal{P}) = \max_{l \in 1 \text{ to } L+1} (C\_time(\mathcal{P}, l)) \tag{3}$$

Note that the above equation is predicting the time for a fixed loop structure with fixed tile sizes. Next we present how to select the tile sizes and loop permutation.

3.4 Tile size and loop permutation selection

Finding efficient tile-sizes for a fixed loop permutation can be formulated as a constrained optimization problem. Our objective is to find the tile sizes such that minimizes the total execution time.

$$\begin{aligned} &\arg\min_{tile-sizes} (TotTime(\mathcal{P})) = \\ &\arg\min_{tile-sizes} (\max_{l \in 1 \text{ to } L+1} (C\_time(\mathcal{P}, l))) \end{aligned} \tag{4}$$

In order to reduce the search space, the sum of data movement for all arrays at each cache level l ($C\_DM(l)$) is constrained to be less than or equal to cache capacity at that level. Let $group\_outer(l)$ denote the outermost tiling loop corresponding to cache level l. The capacity constraint can be expressed as

$$\forall l \in 1 \text{ to } L \sum_{A \in tensors} DM(A, group\_outer(l)) \leq CacheCapacity(l) \tag{5}$$

where L is the number of cache levels.

The tile selection model in Equation (4) relies on an optimistic assumption that reducing the data movement cost corresponding to the most constrained cache level will achieve the best performance. However, the tile sizes obtained by solving this optimization problem only reduces the data movement of the most constrained cache level; the tile-sizes of other cache levels may not be optimal. In real machines, even though the performance is mostly limited by the most constrained cache, the data movement cost of other cache levels also impact the performance. Hence, we modify the previous single level optimization problem to a multi-level optimization problem.

Let T be set of all tile sizes. Let T_l be set of tile sizes such that all tile sizes in T_l affect the data movement at cache level l ($C\_DM(l)$). In other words, varying any $t \in T_i$ will change $C\_DM(l)$ and changing any $t \notin T_i$ wont affect $C\_DM(l)$.

Let j be the most constrained cache level. In other words

$$\forall i \in 1 \text{ to } L+1, (C\_DM(j)/C\_BW(j)) \geq (C\_DM(i)/C\_BW(i)$$

After fixing the tile sizes for j-th cache level, the next constrained cache level can be found using

$$\arg\min_{T - T_j} (\max_{l \in (1 \text{ to } L+1) - T_j} (C\_time(\mathcal{P}, l))) \tag{6}$$

The solution to Equation (6) can be used to identify the second most constraining cache level. This processes can then be repeated for each level of cache.

In order to compute the best permutation, we could iterate over all possible permutations and select the one with best-predicted execution time (Equation (4)). However, this search space grows exponentially with the degree of the tensor/array. Even for a simple example such as 2-level tiled matrix multiplication (Listing 4), there are 362880 (9!) possible permutations. The search space can be reduced by relying on the fact that interleaving tiling loops corresponding to different cache levels are not beneficial. In other words, we only need to consider permutations of tiling loops which correspond to the same cache level. For the matrix multiplication example, this property reduces the search space from 9! to 216 (3! × 3! × 3!). As explained in the overview section, the data reuse at any cache level is dominantly determined solely based on the innermost loop within a set of tiling loops which correspond to the same cache level determines reuse. For the matrix multiplication example, this property further reduce the search space from 216 to 9 (3 × 3 × 3). Let $\mathcal{R}$ represent reduced search space. The final solution is given by

$$final\_solution = \arg\min_{\mathcal{P} \in R} (\arg\min_{tile-sizes} (TotTime(\mathcal{P}))) \tag{7}$$

3.5 Solver

The optimization problem presented in Equation (7) is a non-convex, constrained optimization problem. We use a non-convex, nonlinear programming problem from Couenne[2] (https://projects.coin-or.org/Couenne), released by the COIN-OR (Computational Infrastructure for Operations Research) to solve Equation (7). Couenne (Convex Over and Under Envelope's for Nonlinear Estimation) is a branch and bound algorithm to solve Mixed-Integer Nonlinear Programming

(MINLP) problems of the form:

$$\begin{aligned} &\min f_0(x,y), \\ f_i(x,y) \leq 0, i = 1,2,...,m \\ &x \in R^n, y \in Z^p \end{aligned} \tag{8}$$

where all $f_i(x,y)$ are nonlinear functions.

4 MICRO KERNEL: MAXIMIZE SIMD INSTRUCTION UTILIZATION

Many modern processors include SIMD (vector) instructions to improve parallelism. In order to achieve peak machine throughput, it is important to keep the functional unit busy. Functional units can be kept busy if i) sufficient Instruction Level Parallelism (ILP) is maintained and ii) the memory stalls are avoided/minimized by effectively using the cache.

Let *MaxIssue* be the maximum number of SIMD instructions that can be issued per clock cycle. Let *WordPerVec* be the width of vector instructions. Let *Latency* be the number of clock cycles needed for the instruction to finish all pipeline stages. During each of the clock cycles corresponding to the *Latency*, *MaxIssue* independent instructions have to be issued to keep the pipeline full. Thus $MaxIssue * Latency$ is the minimum number of independent instructions to keep the pipeline full. Since each of these instructions should be independent, the results of these instructions should be kept in distinct registers. Thus the minimum register capacity required is $MaxIssue * Latency * WordPerVec$. BLIS micro-kernel [8] for matrix multiplication follows this design. Our micro-kernels for tensor contraction are based on the BLIS micro-kernel and follows this design.

5 PACKING

A packing routine is a transformation that copies a tile of a tensor into a contiguous buffer. The elements in the buffer are ordered based on the order in which the elements are accessed by the kernel. Thus, consecutive accesses to the tensor are guaranteed to be unit-stride apart. Unit-stride accesses enables usage of efficient load and store SIMD instructions. In addition, packing also provides a tunable mechanism to reduce conflict misses.

5.1 Contiguous Loads and Stores

Efficient use of SIMD instructions in our kernel requires that the input data is stored contiguously in memory in the order that is accessed by the microkernel. Consider the tensor contraction $C[i,j,l] = A[j,i,k] * B[l,k]$. Assume that the innermost loops correspond to dimensions i and l. In the original tensor A and B, the unit-stride access corresponds to dimension k. However, since the innermost loops correspond to dimensions i and l, the data should be packed such that the unit-stride for A is along i and B is along l.

Figure 1 illustrates a simplified version of packing for dense matrix-matrix multiplication (GeMM) micro-kernels. Typical high-performance GeMM micro-kernels perform a set of outer products corresponding to a column vector of A and a row vector of B. Assuming row-major layout, the

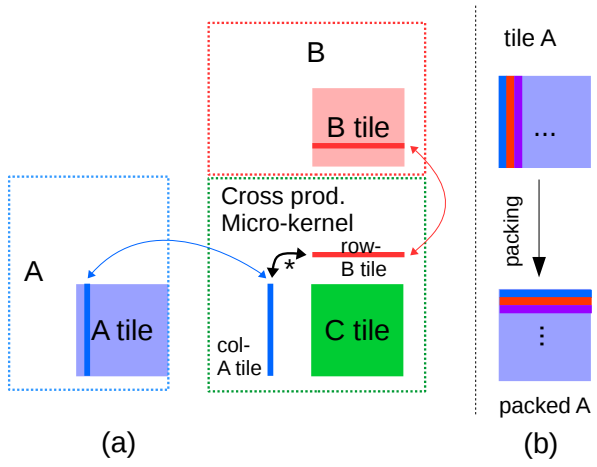

Figure 1: Data packing

```
packCounter = 0
for(int i3 = 0; i3 < Ni; i3+=Ti2)
  for(int k3 = 0; k3 < Nk; k3+=Tk2)
    for(int i2 = 0; i2 < Ti2; i2+=Ti1)
      for(int k2 = 0; k2 < Tk2; k2+=Tk1)
        for(int i1 = 0; i1 < Ti1; i1++)
          for(int k1 = 0; k1 < Tk1; k1++)
            A_Buffer[packCounter++]
            = A[i1+i2+i3][k1+k2+k3]
```

Listing 5: Psuedocode for packing elements of A corresponding to code in Listing 4

elements corresponding to B vector are laid out contiguously in memory. However, the elements of A are not contiguous and hence packing is required. During packing, the columns of A are transposed and placed in *packedA*. The microkernel can then use vector loads to load elements of A using the *packedA* buffer. In addition to efficient loads and stores, packing also helps to reduce TLB misses.

Listing 5 shows the pseudo-code for packing the elements of A corresponding to the 2-level tiled matrix multiplication example (Listing 4)

5.2 Reducing Conflict Misses

One of the major advantages of packing is reduced conflict misses. Typical caches in modern architecture are set-associative. The entire cache is divided into sets and the sets are further sub-divided into lines/ways. A mapping function determines the memory address to set mapping. Within each set, a given memory address can occupy any cache line. In such a design, a memory access can produce conflict misses, where a line in cache is swapped out and replaced even if that line was not the Least Recently Used (LRU) element. By carefully choosing the tile sizes and rely on the fact that the packing routine is designed such that the order in which

data elements are arranged is same as the order that they will be accessed, conflict misses can be avoided.

Note that the packed buffers occupy contiguous regions of memory. Hence, the packed buffers are distributed along all the sets in the cache. Since most caches are not programmable, loading elements of one tensor could evict elements of other tensors. In order to prevent this the number of cache lines each tensor occupies is carefully controlled. For the matrix-multiplication example the number of lines dedicated for A, B and C at cache level l can be computed as

$$\begin{aligned} Line_A &= \lceil DF(A,l)/(NumOfSets(l) * lineSize(l)) \rceil \\ Line_B &= \lceil DF(B,l)/(NumOfSets(l) * lineSize(l)) \rceil \\ Line_C &= \lceil DF(C,l)/(NumOfSets(l) * lineSize(l)) \rceil \end{aligned} \quad (9)$$

$Line_A(l), Line_B(l) and Line_C(l)$ satisfy the constraint $Line_A(l) + Line_B(l) + Line_C(l) \leq Associativity(l)$.

In order understand how packing helps to buffer a particular array in a particular cache level, consider a simplified version of matrix multiplication where arrays A and B are only accessed (accesses to C are ignored). Consider an l level tiling loop corresponding to j. Since A is not indexed by j, A should be buffered at cache level l. During the execution of the j loop, the lines corresponding to A are accessed multiple times. Assuming LRU policy the lines corresponding to A are expected to remain in the cache as they are accessed multiple times. When the cache is full lines corresponding to B have a higher probability of being evicted as they have a lower time stamp. Thus the B elements will be streamed through the cache and A elements will remain stationary.

5.3 Packing Data Movement Model

The packing routine adds additional data movement and computations which could increase the time cost. In tiled execution of a code, the same tile may be packed multiple times. In order to reduce the cost of packing, the packed data must be reused. Due to cache constraints, full reuse of all packed arrays is not possible. The packing cost can be modeled as follows. Assume that A is the only tensor that needs to be packed. Let the IS represent the iteration space (the set of all loops). Let IS_A be a subset of IS which contains all indices used to access A. Assume that the packing is done at the last level of cache (ll). The cost for packing includes the cost to load the data from the main memory and the cost to store the elements to the ll cache.

$$PackCost^{A,buf \in mem}_{mem \to l3} = \prod_{idx \in IS_A} idx \quad (10)$$

Assume the loop order of L3 tiling group is $i_1^{L3}, i_2^{L3}, ..., i_l^{L3} - 1, i_l^{L3}$, and assume i_2, i_l are the only reuse index of A, that is $i_2, i_l \notin IS_a$. If a buffer for packing A is created at L3 level then we have the following:

```
for loop i_1^{L3}
  for loop i_2^{L3}
    ...
      for loop i_l^{L3}
        Packing buffer resides here;
```

Note that the A packing buffer must be filled at each iteration of loop i_2, even if i_2 is a reuse index for A. This means that the total data movement for inside-cache-packing of a given tensor is the product of the tensor size and the ranges of all level of reuse loops above of the packing buffer's resident level. We can describe this scenario as follows:

$$RDX^{L3} = \{i_g^{L3} | i_g \notin IS_A \wedge (\exists i_h \in IS_A)[i_g^{L3} > i_h^{L3}]\} \quad (11)$$

$$PackCost^{A,buf \in L3}_{mem \to L3} = \prod_{idx \in IS_A} idx * \prod_{rdx \in RDX^{L3}} NIter(rdx) \quad (12)$$

Where we define $i_p^{L3} > i_q^{L3}$ to mean that loop i_p^{L3} is above loop i_q^{L3} in L3 tiling group. Additionally, let $NIter(i_p^{L3})$ be the number of iterations of loop i_p^{L3}. Finally, let $Tile(i_p^{L3})$ be the tile size at L3 level for index p. Let N_p be the problem size or global range of index p.

In the previous case the packing buffer is an explicitly allocated block in memory. If the buffer reside in an inner level cache, say L2, then it may not get reuse in the L3 level. This is because in the L2 level, the packing buffer is continually rewriting data to itself and those rewrite may also pollute data in L3.

Therefore, for arbitrary cache L_c,

$$RDX^{L_c} = \{i_g^{L_c} | i_g \notin IS_A \wedge (\exists i_h \in IS_A)[i_g^{L_c} > i_h^{L_c}]\} \quad (13)$$

$$\begin{aligned} PackCost^{A,buf \in L_c}_{mem \to L_c} &= \prod_{idx \in IS_A} idx * \prod_{rdx \in RDX^{L_c}} NIter(rdx) \\ &= \prod_{idx \in IS_A} idx * \prod_{i_p \in RDX^{L_c}} (N_p / Tile(i_p^{L_c})) \end{aligned} \quad (14)$$

A simple combination of the packing model and the computation model stated in Section 4 is added to the packing cost computed here to the DM_i of the computation model. However, from the packing model it is clear that moving the buffer to inner cache will significantly increase the data movement and the number of instructions to be executed. Leaving the packing buffer in memory level will multiply the total required memory. Therefore, it would be the best option to leave the buffers at the L3 level.

Packing in L3 vs. lower levels of cache. Packing at inner levels increases the number of times each data element is packed, which in turn increases the data movement. The number of times each element is packed depends on the tile-level at which the packed-buffer is placed. Since the tile-sizes corresponding to the L3 cache are the highest, our model correctly predicts that the data movement will be lowest for L3 packing. In addition to the data movement cost, packing also requires expensive modulo and division operations. For example, on the Broadwell processor, for the "abcd-aebf-dfce (all 72)" Tensor contraction, L3 packing achieved 43.5 GFLOPS whereas packing at L2 only achieved 19.0 GFLOPS.

5.4 Cache Line Reuse

For our machine model the cache line is the basic unit for moving data between memory hierarchy. Maximum cache line reuse can be achieved by accessing the tensor along the fastest accessing index (the unit stride dimension). We can extend our model to incorporate this as follows.

The packed data automatically obtain the maximum cache line reuse, because the packing order is exactly the order accessed by loop iterations. However the original data do not have this property. To obtain cache line reuse for loading original data, a tiling loop for the fastest index of original tensor layout can be added under the innermost level of packing routine, where tile size is equal to the cache line size. When the original packing routine is not accessing original data layout continuously in fastest index, this added tiling loop will always reduce the total cache lines to be removed, to $1/cacheLineSize$ of original.

5.5 Discussion

The modeling approach imposes a constraint that tile sizes for later levels of the cache must be greater than or equal to the corresponding tile sizes for earlier level caches. If the per-core capacity of an L2 cache is less than the capacity of the private L1 cache, then the generated solution will satisfy the capacity constraints at all levels of cache, and may leave some L1 capacity unused. In such a situation, we do not see any way of fully utilizing L1 capacity while not exceeding L2 capacity. We note that the modeling approach assumes an inclusive multi-level cache - exclusive caches can also be handled by using the sum of L1+L2 capacities as the modeled L2 capacity in the modeling.

6 EXPERIMENTS

This section presents experimental results. We conducted experiments on two target platforms: an Intel Core i7-6700K dual socket 28-core Broadwell processor and an Intel Xeon CPU E5-2680 v4 single socket quad core Skylake processor. We compared our implementation with two tensor contraction libraries, TBLIS [9] and TCL[13]. TBLIS is a BLIS [15] based library to perform tensor contraction without explicit transpose. TCL is a library for computing tensor contractions using explicit transpose and high-performance GEMM. TCL uses the HPTT[14] library to perform the transposition, and either the Intel MKL library [17] or BLIS for GEMM. ACMTC denotes our approach. We used the GNU GCC 7.3.0 compiler with -O3 and –std=c99 flags. For TCL-MKL, we used MKL 2018 to perform BLAS operations. We perform comprison with both versions of TCL: with the TCL-MKL version because it is the higher performing version, as well as TCL-BLIS because it reprrsents a better "apples-to-apples" comparison with ACMTC since it also uses the same BLIS micro-kernel as TCL-BLIS. Table 4 lists all the information of contraction examples we used, from the TCCG benchmarks [13].

This paper focuses on modeling data movement at the different levels of the memory hierarchy for sequential multi-level tiled execution of tensor contractions. The model can be extended for parallel multicore execution of a tensor contraction, where different cores execute adjacent tiles along a parallelizable dimension of the iteration space. The handling of shared levels of cache will depend on whether the tile data footprints of the arrays are the same across the cores or disjoint: for the disjoint data slices the capacity must be partitioned. The development of a model-driven tiled code generation strategy for parallel execution of a tensor contraction is still under development. However, we carried out experiments for parallel execution in the simpler scenario of "batched" tensor contractions, where a batch of independent tensor contractions on disjoint data needs to be performed. With this scenario, since all data processed by the different cores is completely disjoint, we simply model the shared-level L3 cache as having a capacity of $\frac{1}{14}$ the per-socket L3 caches in the i7 processor and $\frac{1}{14}$ of the L3 cache capacity for the quad-core processor.

6.1 Assessment of Data Movement Model

In this section, we assess the accuracy of the data movement prediction model by comparing the predicted volume of data movement with measured cache misses obtained using PAPI on the Intel i7-4770K Broadwell processor. We select four tensor contractions as the test cases. The label for each test case specifies the order of indices in the output and input tensors. For example, the label abcdef-degb-gfac represents the contraction C[a,b,c,d,e,f] = A[d,e,g,b]*B[g,f,a,c]. The number of tensor dimension varies from four to seven. Each example maps to one of the cases in (small A, B, large C), (small A, large B, C), (small C, large A, B), (large A, B, C). The table 2 shows the measured cache line misses and the predict data movement in cache lines during the computation phase. Our predicted data movement is close to the actual data movement.

Figure 2 compares cache misses for ACMTC to TBLIS and TCL for the four representative tensor contraction expressions (one each from CCSD, CCSD(T), contractions involving tensor multiplication and two-electron integrals transform). In order to obtain accurate cache miss data, we disabled the hardware prefetcher. The combined data movement of our approach is consistently lower than all the other approaches.

6.2 Performance Evaluation

We created a set of micro-benchmarks to measure the bandwidths of the machines at the different levels. Each micro-benchmark consists of a sequence of Load-FMA-Store instructions on a continuous memory block of a given size with no reuse. We start running the micro-benchmark from a small memory block whose size is less than half of the L1 cache and increase the size of memory to be accessed exponentially, till it is close to two times of the size of L3 cache. We recorded the total accessed data amount and time needed, and the bandwidth for that size of data is computed by dividing time

		Computation, Total		Cache Misses, Computation		Operation Intensity, Total		Operation Intensity, Computation	
Benchmark	**Cache**	**Actual**	**Predict**	**Actual**	**Predict**	**Actual**	**Predict**	**Actual**	**Predict**
abcdef-degb-gfac	**L1**	8.26E+06	8.13E+06	8.06E+06	7.84E+06	219.43	222.84	224.79	231.23
	L2	7.08E+06	6.80E+06	6.96E+06	6.51E+06	256.02	266.41	260.48	278.48
	L3	4.80E+06	5.05E+06	4.78E+06	4.76E+06	377.38	358.77	379.18	381.02
abcd-aebf-dfce	**L1**	2.64E+09	2.47E+09	2.50E+09	2.44E+09	105.68	112.73	111.37	114.28
	L2	3.29E+08	3.01E+08	2.94E+08	2.68E+08	846.04	924.52	949.12	1039.8
	L3	1.32E+08	1.34E+08	1.04E+08	1.01E+08	2103.2	2079.22	2676.06	2769.79
abcde-efbca-fd	**L1**	7.67E+07	3.30E+07	2.35E+07	2.36E+07	89.66	208.28	293.36	291.6
	L2	7.33E+07	3.10E+07	2.17E+07	2.16E+07	93.87	221.86	316.64	318.94
	L3	2.86E+07	3.07E+07	2.15E+07	2.12E+07	240.55	224.3	319.42	324
abcd-ea-ebcd	**L1**	4.36E+07	7.85E+07	3.49E+07	1.34E+07	88.80	49.32	110.92	288.00
	L2	1.48E+07	2.18E+07	7.06E+06	6.83E+06	261.70	177.16	548.39	566.96
	L3	6.78E+06	1.35E+07	6.77E+06	6.72E+06	571.15	285.73	571.76	575.94

Table 2: Measured Cache Misses and Model Predicted Data Movement

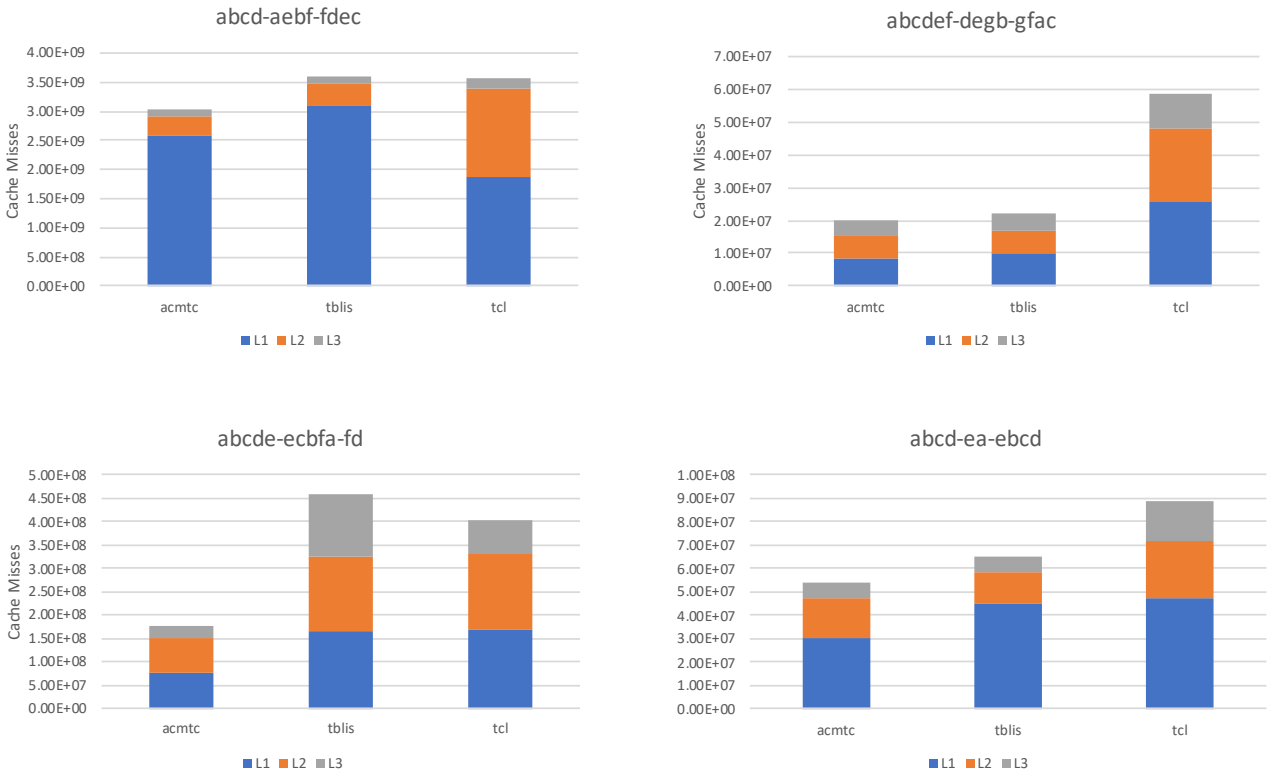

Figure 2: Measured Cache Misses for ACMTC, TBLIS, and TCL

by the volume of the accessed data. When the amount of data accessed in micro-benchmark is closest but smaller than some level of cache, it can fit into that level of cache, and the bandwidth of this amount of data could be seen as the bandwidth of that level of cache. We did not use STREAM benchmark or PMBW benchmark as the bandwidth reported by these benchmarks reflects the maximum achievable bandwidth under the assumption that a load/store instruction can be issued every clock cycle. Since tensor contractions require other instructions such as FMA, it is not feasible to issue load/store instructions every clock cycle. Using the PMBW bandwidth reduced the quality of our model. For example, for abcdef-dega-gfbc TC, the performance achieved

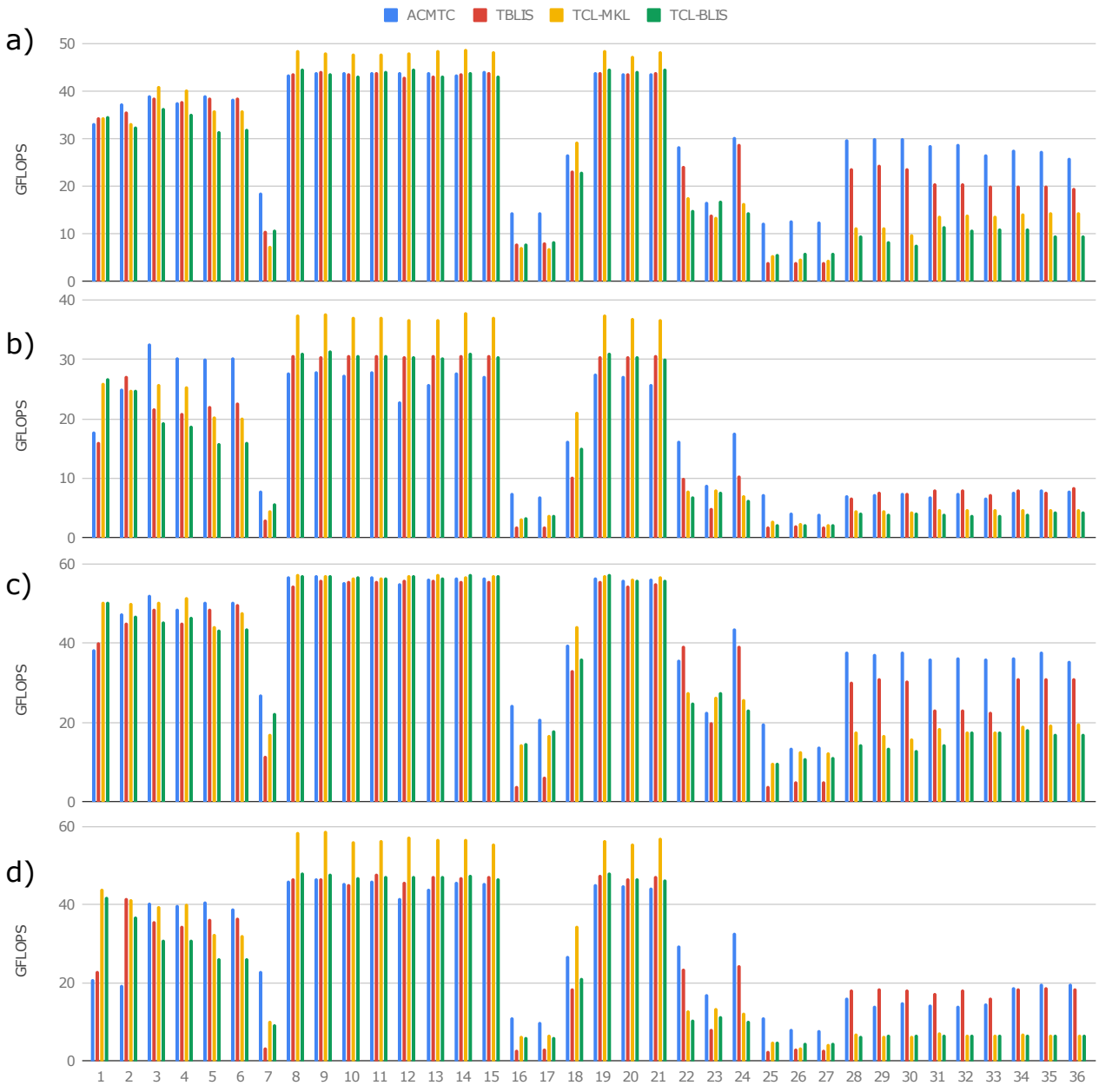

Figure 3: Tensor Contraction Performance Comparison on TCCG benchmarks (a) Broadwell single core, (b) Broadwell multi-core, (c) Skylake single core, (d) Skylake multi-core

using our microbenchmark bandwidth was 29 GFLOPS; using bandwidths reported by the PMBW benchmark reduced the performance to 25 GFLOPS.

The measured bandwidths for the Intel Core i7-6700K processor and the Intel Xeon E5-2680 v4 processor measured by the micro-benchmark is listed in the Table 3.

As shown in 3 ACMTC achieves higher performance for single core on all benchmarks when compared to TBLIS. We outperformed TCL in most cases.

For Broadwell architecture, as shown in Figure 3, the geometric mean of the speedup is 1.25x versus TBLIS, 1.41x versus TCL-MKL, and 1.51x versus TCL-TBLIS. On Skylake architecture, as shown in Figure 3, the geometric mean

Measured Bandwidth (byte/cycle)	L1	L2	L3	Memory
i7-6700K Skylake	19.36	18.32	12.8	6.4
Xeon E5-2680 Broadwell	25.28	19.68	11.44	6.08

Table 3: Measured Bandwidth on Skylake and Broadwell

	Expression	Problem size
1	ab-acd-dbc	a:312 b:312 c:312 d:312
2	ab-cad-dcb	a:312 b:312 c:312 d:312
3	abc-acd-db	a:312 b:312 c:312 d:312
4	abc-ad-bdc	a:312 b:312 c:312 d:312
5	abc-adc-bd	a:312 b:312 c:312 d:312
6	abc-adc-db	a:312 b:312 c:312 d:312
7	abc-bda-dc	a:312 b:312 c:24 d:312
8	abcd-aebf-dfce	a:72 b:72 c:72 d:72 e:72 f:72
9	abcd-aebf-fdec	a:72 b:72 c:72 d:72 e:72 f:72
10	abcd-aecf-bfde	a:72 b:72 c:72 d:72 e:72 f:72
11	abcd-aecf-fbed	a:72 b:72 c:72 d:72 e:72 f:72
12	abcd-aedf-bfce	a:72 b:72 c:72 d:72 e:72 f:72
13	abcd-aedf-fbec	a:72 b:72 c:72 d:72 e:72 f:72
14	abcd-aefb-fdce	a:72 b:72 c:72 d:72 e:72 f:72
15	abcd-aefc-fbed	a:72 b:72 c:72 d:72 e:72 f:72
16	abcd-dbea-ec	a:72 b:72 c:72 d:72 e:72 f:72
17	abcd-deca-be	a:72 b:72 c:72 d:72 e:72 f:72
18	abcd-ea-ebcd	a:72 b:72 c:72 d:72 e:72
19	abcd-eafb-fdec	a:72 b:72 c:72 d:72 e:72 f:72
20	abcd-eafc-bfde	a:72 b:72 c:72 d:72 e:72 f:72
21	abcd-eafd-fbec	a:72 b:72 c:72 d:72 e:72 f:72
22	abcd-eb-aecd	a:72 b:72 c:72 d:72 e:72
23	abcd-ebad-ce	a:72 b:72 c:24 d:72 e:72
24	abcd-ec-abed	a:72 b:72 c:72 d:72 e:72
25	abcde-ecbfa-fd	a:48 b:32 c:32 d:24 e:48 f:48
26	abcde-efbad-cf	a:48 b:32 c:24 d:32 e:48 f:32
27	abcde-efcad-bf	a:48 b:24 c:32 d:32 e:48 f:32
28	abcdef-dega-gfbc	a:24 b:16 c:16 d:24 e:16 f:16 g:24
29	abcdef-degb-gfac	a:24 b:16 c:16 d:24 e:16 f:16 g:24
30	abcdef-degc-gfab	a:24 b:16 c:16 d:24 e:16 f:16 g:24
31	abcdef-dfga-gebc	a:24 b:16 c:16 d:24 e:16 f:16 g:24
32	abcdef-dfgb-geac	a:24 b:16 c:16 d:24 e:16 f:16 g:24
33	abcdef-dfgc-geab	a:24 b:16 c:16 d:24 e:16 f:16 g:24
34	abcdef-efga-gdbc	a:24 b:16 c:16 d:24 e:16 f:16 g:24
35	abcdef-efgb-gdac	a:24 b:16 c:16 d:24 e:16 f:16 g:24
36	abcdef-efgc-gdab	a:24 b:16 c:16 d:24 e:16 f:16 g:24

Table 4: Tensor Contraction Benchmarks for Performance Evaluation

speedup is 1.34x versus TBLIS, 1.27x versus TCL-MKL, and 1.34x versus TCL-BLIS respectively.

We also conducted experiments for the multi-core environment for a "batched" contraction scenario where a number of identical contractions are performed on different operands. We modeled the shared L3 cache as logically divided into equal-sized parts for each core on a socket. For each core, the same tensor contraction benchmark was launched on each core simultaneously on independent data. An MPI barrier was set at the beginning and end of the computation. The average performance per core is shown in (b) and (d) of figure 3 in GFLOPS. Overall, on the Broadwell CPU, the geometric mean of speed up is 1.25x versus TBLIS, 1.21x versus TCL-MKL, and 1.38x versus TCL-BLIS. On the Skylake CPU, the geometric mean of speedup is 1.23x versus TBLIS, 1.31 versus TCL-MKL, and 1.47 versus TCL-BLIS. We observe that ACMTC as well as TBLIS and TCL achieve lower per-core performance for the multi-core scenario than the single-core case. A significant reason for this is the lower per-core capacity available in the shared L3 cache. Further, we note that the speedup of ACMTC over TBLIS and TCL for the multi-core case is lower for some of the benchmarks and higher for others. A significant reason appears to be differences in the cross-thread interference in the shared L3 cache. For example, for benchmark No.1, where ACMTC suffers the greatest loss of the performance relative to TCL for the multi-core scenario, the L3 miss count increases from 6.2 millio misses per core to 41 million misses per core on 28 threads for ACMTC, but only rises from around 9 million to 15 million misses for the TCL-MKL and TCL-BLIS versions.

6.3 Discussion

Time prediction. Even though the primary focus of our modeling is to aid the choice of tile-loop permutation and tile sizes, our model can also be used to predict the execution time. The error rate of our time prediction model was less than 10% in most cases. This time prediction model can be used to evaluate different architectural choices. As an example, consider the contraction: abcdef-degb-gfac (problem size a to g: 24,16,16,24,16,16,24). On an Intel Xeon E5-2680 v4 processor (Broadwell), with 128 GB RAM, our data movement model shows that the performance of the above TC is bottlenecked by the Memory to L3 bandwidth. From an architectural standpoint, there are two main ways to alleviate this bottleneck: i) increase Memory bandwidth ii) increase L3 cache size. Our time prediction model predicts that if the Memory bandwidth is increased by 5%, the performance (GFLOPS) will also increase by 5%. It also predicts that increasing memory bandwidth beyond 21% will change the bottleneck to L1-to-Register bandwidth. On the other hand, if we increase the L3 cache size, our model predicts that the performance will not improve.

7 RELATED WORK

There has been extensive prior work on loop optimization. Polyhedral compilers [1, 3, 6, 16] have developed very powerful loop transformation strategies for tiling complex imperfectly nested affine loop computations. Tensor contractions are special cases of affine loop computations and therefore polyhedral compilers can tile code for arbitrary tensor contractions. However the cost models used for guiding choice of loop transformations in polyhedral compilers are constrained to be linear functions, while the tile size optimization problem is inherently a nonlinear optimization problem, as discussed in detail in this paper. The linear cost models used internally in polyhedral compilers are too imprecise to effectively choose the best among the exponential number of permutations of the tiled loops.

All previously proposed performance modeling approaches in compilers either suffer from imprecision or an exponential blow-up in the number of cases that have to be evaluated

in optimizing the tiling configurations (permutations of the tiling loops) and tile size selection.

The topic of analytical modeling for tile size optimization has been addressed by a number of prior research efforts [4, 10, 11, 18]. However, previous modeling approaches suffer from one or more of the following shortcomings: a) they use a model of nested tiles that are optimized in some fixed sequence (in contrast to our approach of solving the multi-level tile size selection problem in a coupled fashion); b) they do not model inter-tile reuse. Finally, prior efforts on tile size optimization generally compare performance or speedup of the optimized tiled code with untiled baseline codes; comparisons with the best available manually optimized code or code from state-of-the-art libraries are rarely done. In contrast, we demonstrate the effectiveness of the modeling approach over an extensive public benchmark suite for tensor contractions, by comparing performance with the best-known implementations for those contractions from state-of-the-art libraries.

8 CONCLUSION

In this paper we have presented a new methodology for multi-level tile-size optimization for a class of nested loop tensor computations. It is based on observations that enable significant reduction of the search space and an approach to analytical characterization of data volume at each level of a multi-level storage hierarchy along with solution using a constrained optimization solver. The effectiveness of the modeling and optimization approach was demonstrated over a large set of tensor contractions. The approach is more broadly applicable and is being extended to optimize machine learning kernels.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback. This work was supported in part by the U.S. National Science Foundation through awards 1816793, 1513120, and CCF-1619303, and by the Louisiana Board of Regents through the award LEQSF (2016-19)-RD-B-03.

REFERENCES

[1] Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE.

[2] Pietro Belotti. 2009. *Couenne: a users manual*. Technical Report. Technical report, Lehigh University.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. PLUTO: A Practical and Fully Automatic Polyhedral Program Optimization System. In *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*.

[4] Stephanie Coleman and Kathryn S. McKinley. 1995. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, 279–290.

[5] T Daniel Crawford and Henry F Schaefer III. 2000. An introduction to coupled cluster theory for computational chemists. *Reviews in computational chemistry* (2000), 33–136.

[6] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* 21, 5 (1992), 313–347.

[7] Kazushige Goto and Robert A Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 12.

[8] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software (TOMS)* 43, 2 (2016), 12.

[9] Devin A Matthews. 2018. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing* 40, 1 (2018), C1–C24.

[10] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. 2008. Positivity, Posynomials and Tile Size Selection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Article 55, 12 pages.

[11] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J Ramanujam, P Sadayappan, and Vivek Sarkar. 2012. Analytical bounds for optimal tile size selection. In *International Conference on Compiler Construction*. Springer, 101–121.

[12] Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1049–1059.

[13] Paul Springer and Paolo Bientinesi. 2016. Design of a High-Performance GEMM-like Tensor-Tensor Multiplication. *CoRR* (2016). arXiv:cs.MS, cs.PF/1607.00145 http://arxiv.org/abs/1607.00145

[14] Paul Springer, Tong Su, and Paolo Bientinesi. 2017. HPTT: A High-Performance Tensor Transposition C++ Library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2017)*. ACM, New York, NY, USA, 56–62. https://doi.org/10.1145/3091966.3091968

[15] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 14.

[16] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[17] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 167–188.

[18] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. 2010. Automatic Creation of Tile Size Selection Models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, 190–199.